Sprachen für Computer (2)

Diesmal sollen uns bei unserem Überblick über die Welt der Programmiersprachen die modernen Sprachen interessieren. Sie erfahren, was Ada kann und was man sich unter Modula oder Lisp vorzustellen hat.

Die »klassischen« alten Programmiersprachen wie Fortran, Cobol oder Algol haben viel mehr miteinander gemeinsam, als man beim oberflächlichen Betrachten von Programmen in einer dieser Sprachen erwarten würde. So können zum Beispiel nur numerische Datentypen verwendet werden, eine Einschränkung, in der sich die unzulänglichen Hardwaremöglichkeiten der fünfziger Jahre wiederspiegeln. Tatsächlich war Rechenzeit und Speicherplatz damals Mangelware, und niemand dachte auch nur im entferntesten daran, die teure Rechnerkapazität für andere Dinge als schwierige numerische Berechnungen einzusetzen. Das führte dann später in den sechziger und siebziger Jahren zur Unsitte, rein numerisch orientierte Sprachen wie Fortran schließlich für alle möglichen Zwecke einzusetzen. Die Sprache war halt einmal vorhanden, Rechenkapazität verfügbar und so wurden sogar ganze Textverarbeitungsprogramme in Fortran geschrieben, Buchstabe für Buchstabe fein säuberlich als Zahlenwert codiert und in einem numerischen Datenfeld abgelegt.

Da war eine Sprache wie Basic, das 1965 am Dartmouth College entwickelt worden ist, schon ein gewisser Fortschritt. Basic bietet neben numerischen Datentypen auch noch Strings, also Zeichenketten, als Datentyp an und stellt auch spezielle Operatoren dafür zur Verfügung. Tatsächlich ist mit der Möglichkeit, numerische Daten und Texte zu bearbeiten, schon ein großer Teil der (damals) denkbaren Computer-Anwendungen abgedeckt. Moderne Programmiersprachen stellen aber noch weitaus mehr Komfort zur Verfügung. Es können beliebige eigene Datentypen definiert werden, es gibt sogenannte »Verbundvariable«, zusammengesetzt aus mehreren Variablen verschiedener Datentypen, und schließlich können auch komplexe Datenstrukturen wie verkettete Listen oder Suchbäume einfach dargestellt werden. Die erste Sprache, die diese Möglichkeiten konsequent verwirklichte, war Pascal, auf das schon im ersten Teil ausführlicher eingegangen wurde. Im folgenden wollen wir uns etwas näher mit den Nachfolgern von Pascal beschäftigen, aber auch auf völlig andersartige Konzepte eingehen.

Pascal in Kurzform: C

Die Entwicklung der Sprache C ist eng verbunden mit der Geschichte des 16-Bit-Betriebssystems Unix, das selbst vollständig in C geschrieben ist. Der Name dieser Sprache hat eine verblüffend einfache Herkunft. Zur Programmier-Unterstützung auf Minicomputern von Digital Equipment wurden um 1970 Spezialsprachen entwickelt, die einfach nach dem Alphabet die Bezeichnungen A und B erhielten. Die B-Sprache wurde 1971 dazu benutzt, Unix, das damals noch in Assembler geschrieben war, auf einfache Art und Weise auf andere Computer zu übertragen. Schließlich erkannte Dennis Ritchie, ein Programmierer bei Bell Labs, die Fähigkeiten der Sprache B, erweiterte und verfeinerte sie und nannte das Resultat C.

Kurz darauf (1973) wurde Unix auf C umgeschrieben — das erste Betriebssystem, das in einer höheren Programmiersprache abgefaßt war. Das war nur möglich, weil C ein strukturiertes Sprachkonzept bei gleichzeitiger größtmöglicher Effizienz der Programme zur Verfügung stellt. In C geschriebene Programme sind einerseits sehr kompakt (geringer Speicherbedarf), andererseits sehr schnell (typischerweise etwa 50 mal so schnell wie Basic).

Programmierung in C besteht im wesentlichen im Schreiben von Funktionen, die jeweils Teilbereiche des Problems lösen. Die Gesamtlösung ergibt sich dann durch zweckmäßigen Einsatz dieser Funktionen. Dementsprechend ist C einerseits stark an Pascal angelehnt, verfügt andererseits aber über zusätzliche Befehle und Operatoren, die der verbesserten Ausnutzung der vorhandenen Hardware (= des Prozessors) dienen. So bietet C spezielle, an die Assembler-Programmierung angelehnte Befehle zum Inkrementieren und Dekrementieren (erhöhen/erniedrigen) von Variablenwerten und als einzige der verbreiteten höheren Programmiersprachen die Option, Variable als Register-Variable zu deklarieren. Das bedeutet, daß solche Variablen nach Möglichkeiten in internen Prozessor-Registern gehalten werden, was einen wesentlich schnelleren Zugriff als bei den normalerweise verwendeten Speichervariablen ermöglicht. An dieser Stelle merken Sie wahrscheinlich schon, daß bei der Entwicklung der Sprache nicht unbedingt an den 6502-Prozessor gedacht wurde, der mit seinen bescheidenen drei Registern (A,X,Y) in dieser Beziehung nicht gerade überwältigende Möglichkeiten bietet. Allerdings würde sich als Register-Ersatz die Zero-Page anbieten, bei der der Zugriff ja auch etwas schneller ist. Andererseits wurde C auch nicht gerade für 64-Bit-Superprozessoren konzipiert. Derartige in Großrechnern verwendete Prozessoren verfügen in der Regel über eine sehr große Zahl von Registern, und ein guter Fortran-Compiler für so einen Großrechner verwendet natürlich auch diese Register für die Programmoptimierung. So wird C in der Regel auf 16- oder 32-Bit-Maschinen eingesetzt. Bei den 8-Bit-Prozessoren ist C praktisch nur für den Z80 verbreitet (unter CP/M), allerdings soll eine Version für den C 64 in Vorbereitung sein. Wir werden Sie im 64'er Magazin darüber auf dem laufenden halten.

Modulare Programmierung: Modula

Das Grundkonzept aller modernen Programmiersprachen heißt Modularisierung der Programmentwicklung. Damit ist gemeint, Probleme nicht mit immensem Aufwand in umfangreichen Programmen zu lösen (die ja für jedes Problem wieder

neu geschrieben werden müssen), sondern Teilaspekte des Problems in möglichst allgemeiner Form in eigenständigen Teilprogrammen, eben den sogenannten Modulen, zu lösen. Der Vorteil dieser Art der Software-Entwicklung liegt auf der Hand: Ein Teilproblem muß nur ein einziges Mal gelöst werden; tritt das gleiche Teilproblem irgendwann einmal wieder bei der Programmierung auf, kann man auf das fertige Modul zurückgreifen. Außerdem ist es natürlich zumeist viel einfacher, mehrere kleinere Probleme zu lösen als ein großes. Bei konsequentem Einsatz der »modularisierten Programmierung« lassen sich Programme somit auf lange Sicht wesentlich rationeller entwickeln.

Allerdings gibt es einige Voraussetzungen für die Modularisierung: Die einzelnen Module müssen über genau definierte »Schnittstellen« nach »außen« verfügen, damit sie zweckmäßig ausgewählt und eingesetzt werden können. Andererseits darf aber das »Innenleben« der Module auf keinen Fall irgendwelche Dinge »außen« beeinflussen. Basic ist daher zum Beispiel sehr schlecht für diese Art der Programmierung geeignet: Unterprogramme haben keine Namen, sondern werden über Zeilennummern aufgerufen (schlechte Dokumentation), es gibt keine Parameterübergabe, also keine feste Schnittstelle, und schließlich ist jede Variable, die im Unterprogramm verändert wird, anschließend auch im Hauptprogramm verändert.

Modula, entwickelt von Professor N. Wirth, dem Schöpfer der Sprache Pascal, ist eine der ersten Sprachen, die das Konzept der Modularisierung, das zum Teil ja auch schon in Pascal enthalten ist, konsequent durchführt.

Jedes Modula-Programm ist selbst ein Modul und kann dementsprechend andere Programme als Untermodule verwenden. Dabei gibt es keinen Unterschied zwischen den Standard-Modulen, die die Sprache dem Benutzer zur Verfügung stellt und selbstgeschriebenen (Programm-)Modulen. Jedes Modul ist in sich vollständig abgeschlossen, daß heißt Vereinbarungen über Datentypen, Funktionen und Variable gelten nur innerhalb des Moduls. Sollen die in einem Modul getroffenen Definitionen anderen Modulen zugänglich gemacht werden, dann müssen sie vom Modul »exportiert«, von den anderen Modulen »importiert« werden. Damit werden alle unerwünschten Nebeneffekte, wie sie etwa bei Basic-Unterprogrammen häufig auftreten können, vollständig ausgeschlossen.

Selbstverständlich sind die einzelnen Module intern ebenfalls völlig strukturiert aufgebaut. Den Befehl GOTO sollte man als Modula-Programmierer schnellstens vergessen; dafür stehen die von Pascal gewohnten Schleifenstrukturen (FOR...TO... DO...END, REPEAT...UNTIL, WHILE...END) zur Verfügung, zusätzlich noch die LOOP...EXIT...END-Schleife, die alle Befehle zwischen LOOP und END so lange durchläuft, bis die bei EXIT angegebene Bedingung erfüllt ist. Diese Schleifenstrukturen sind übrigens ohne Ausnahme auch im Commodore 3.5-Basic des C 16, oder im 7.0-Basic des C 128 vorhanden, ein sicheres Zeichen, daß sich das Konzept der strukturierten Programmierung auch im Bereich der Basic-Heimcomputer immer mehr durchsetzt. Leider ist Modula noch nicht für Commodore-Computer erhältlich.

Die Krönung: Ada

Den bisherigen Höhepunkt bei der Entwicklung moderner, modularer Programmiersprachen stellt ohne Zweifel die Sprache Ada dar.

Die Sprache hat ihren Namen nach der Countess Augasta Ada Lovelace, die sich als erste Frau bereits im vorigen Jahrhundert (!) mit Algorithmen und Rechenmaschinen beschäftigte. Bekannt wurde sie in erster Linie, indem sie ein »Rechenkalkül« für die von Charles Babbage entworfene mechanische Rechenmaschine entwarf. Dieses Kalkül kann man im weitesten Sinne als den ersten Algorithmus für eine Maschine bezeichnen.

Ada wurde erst in jüngster Zeit im Auftrag des amerikanischen Verteidigungsministeriums entwickelt und baut ebenso wie C und Modula auf dem Konzept von Pascal auf. Die Sprache ist ebenso wie Modula streng modular und strukturiert aufgebaut, geht aber von den Fähigkeiten weit über Modula hinaus.

Wie in Modula gehören praktisch alle Funktionen, insbesondere auch die Ein-/Ausgabe-Operationen nicht direkt zur Sprache, sondern werden in besonderen Modulen, den sogenannten Packages (Pakete) bereitgehalten. Das hat den Vorteil, daß der Anwender im Bedarfsfalle fast alle Funktionen selber neu schreiben kann, wenn ihm die Standard-Funktionen nicht gefallen. Alle Packages, ob selbstgeschrieben oder vom System bereitgestellt, müssen ins Programm, das seinerseit ein Package darstellt, importiert werden. Dies geschieht sehr einfach durch Angabe des entsprechenden Paket-Namens. Um zum Beispiel die Standard-Routinen für die Ein- und Ausgabe von Texten zu importieren, verwendet man die Anweisung:

```
TEXT_IO IS PACKAGE
STANDARD_TEXT_IO
```

Durch diese Angabe werden die im Modul STANDARD_TEXT_IO enthaltenen Funktionen dem Programm-Modul unter dem Namen TEXT_IO bekanntgemacht. Das Standard-Modul enthält beispielsweise die Funktionen WRITE und READ zum Drucken und Einlesen von Daten.

Durch die Anweisung USE TEXT_IO werden dann bei Bedarf (wenn WRITE oder READ im Programm vorkommt) die entsprechenden Funktionen aus dem STANDARD_TEXT_IO-Modul (das ja innerhalb des Programms den Namen TEXT_IO erhalten hat) entnommen und ins Programm eingefügt. Will man aber an einer bestimmten Stelle nicht die Standardfunktion verwenden, sondern eine selbstgeschriebene, dann braucht man das eigene Modul nur mit

```
MEIN_TEXT IS PACKAGE
EIGENE_TEXT_IO
```

anzumelden und im Programm mit

```
USE MEIN_TEXT
```

auf die selbstgeschriebenen Routinen umzuschalten. Dabei wird die Ada-Fähigkeit des »Überladens« von Funktionen verwendet, was nichts anderes bedeutet, als daß ein- und dasselbe Schlüsselwort (oder Zeichen) je nach Kontext grundverschiedene Funktionen bezeichnet. Nach Umschaltung mittels »USE« heißt die Ausgabefunktion in unserem Beispiel zwar immer noch »WRITE«, es wird aber die selbstgeschriebene Funktion aufgerufen und nicht die Standard-Funktion.

Natürlich kann man diese Technik auf alle möglichen Bereiche ausdehnen. Damit wird zusammen mit der Möglichkeit sich eigene Datentypen, wie in Pascal oder Modula, zu definieren, ein sehr hoher Grad an Flexibilität gewährleistet. Zum Beispiel kann man sich bei Bedarf einen Datentyp COMPLEX für komplexe Zahlen definieren und die Operatoren »+«, »-«, »*« und »/« durch Überladen auch auf diesen neuen Datentyp anwenden.

Zwei weitere Aspekte von Ada sind noch besonders erwähnens-

wert: Zum einen existieren sehr komfortable Möglichkeiten, Ausnahmesituationen im Programm (sogenannte Exceptions) ohne Abbruch unter Kontrolle zu bringen. In der Regel dient das zum Abfangen von Fehlerbedingungen, die innerhalb eines Programms auftauchen können (in etwa vergleichbar mit »ON ERROR« oder »TRAP« in Basic). Eine solche Fehlerbehandlung ist bei Compilersprachen bislang sehr selten, für die Zwecke des amerikanischen Verteidigungsministeriums aber natürlich unabdingbar. Man stelle sich nur vor, in kritischen Situationen würden Radarstationen keine Informationen mehr weitergeben wegen eines »DIVISION BY ZERO ERROR« ...

Der zweite zusätzliche Aspekt von Ada ist die Fähigkeit, parallele Prozesse automatisch durchführen zu können (Multitasking). Damit ist gemeint, daß bestimmte Operationen gleichzeitig ablaufen können (sofern der Computer mit mehreren Prozessoren ausgestattet ist, sonst ergibt sich natürlich nur eine »Pseudo-Gleichzeitigkeit«). Das bringt natürlich eine unter Umständen immense Erhöhung der Verarbeitungsgeschwindigkeit, allerdings auf Kosten eines hohen Hardwareaufwandes.

Wer sich etwas näher mit Ada beschäftigen will und einen C 64 besitzt, der findet im Ada-Trainingskurs von Data Becker (siehe Test in dieser Ausgabe) einen brauchbaren Einstieg.

Es gibt ganz grob eingeteilt zwei grundverschiedene Linien unter den Programmiersprachen.

Da sind einmal die sogenannten Anweisungs-Sprachen, zum anderen die funktionalen Sprachen. Zu den Anweisungs-Srachen gehören alle »klassischen« Programmiersprachen einschließlich Basic, daneben natürlich Pascal, Modula, Ada und C. Alle diesen Sprachen sind von Ihrer Struktur her mehr oder weniger stark an die Hardware-Organisation der heutigen Computer angelehnt (Befehl holen — Befehl ausführen — Programmzeiger auf nächsten Befehl setzen). Funktionale Sprachen verlangen demgegenüber nicht die Ausführung spezieller, einzelner Befehle, sondern beschreiben Strukturen und bilden neue Strukturen.

Vertreter dieser Sprachen sind zum Beispiel Lisp (eine listenverarbeitende Sprache), Snobol (eine symbolische Simulations-Sprache) und eine Unzahl kleiner, außerhalb der Universitäten völlig unbekannten Spezialsprachen.

Lisp ist sicherlich der bekannteste Vertreter dieser Sprachenklasse. Für den Laien besteht Lisp in erster Linie aus einer Unzahl von öffnenden und schließenden Klammern, ab und zu findet sich dazwischen auch mal ein anderes Zeichen. Die Ursache dafür ist die funktionale Struktur der Sprache. Es gibt in Lisp keine Befehle, sondern nur Funktionen, die aus primitivsten Grundelementen sehr komplexe Strukturen aufbauen können. Ausgangspunkt ist immer die kleinste, unteilbare Lisp-Struktur, das Atom. Ein Atom kann eigentlich alles sein, was man über eine Tastatur in einen Computer hineinbekommt. Mit (ATOM APFEL) wird ein »DING« namens Apfel definiert, daß weder interne Struktur noch Beziehungen zu irgendwelchen anderen Daten hat. Diese Beziehungen zu anderen Daten werden erst im folgenden durch Funktionen definiert. Wesentliches Element von Lisp (List Processing Language) ist die Fähigkeit, Listen zu bilden. Eine Liste ist eine geordnete Aufzählung von Elementen. Diese Elemente sind im einfachsten Fall Atome, es kann sich dabei aber auch um andere Listen oder gar um Funktionen handeln. Mit

(SETQ OBST (APFEL BIRNE KIRSCHE))

wird eine Liste OBST definiert, die aus den Atomen Apfel, Birne und Kirsche besteht (die natürlich vorher als Atome vereinbart worden sind).

Funktionale Sprachen

Es gibt jetzt eine Menge vordefinierter Funktionen, um diese Listen weiter zu verarbeiten. Es können einzelne Elemente oder ganze Listen zu neuen Listen umgewandelt werden, es können Bedingungen festgelegt werden, unter denen das geschieht und so fort. Dabei können sehr komplexe Strukturen, regelrechte logische Netzwerke, entstehen.

Dadurch wird in gewisser Weise der in erster Linie assoziativ arbeitende Mechanismus des menschlichen Denkens viel besser simuliert, als durch schrittweise Abarbeitung von Befehlen.

Lisp wird denn auch mit durchaus beachtenswerten Erfolgen bei praktisch allen KI-(künstliche Intelligenz-)Projekten eingesetzt. Die Sprache liefert vor allem auf den Gebieten der symbolischen Datenverarbeitung (nicht-numerische Mathematik), der Erkennung von Mustern und dem logischen Folgern gute Ergebnisse.

Für den C 64 ist Lisp nicht erhältlich, jedoch ist mit Logo eine andere Sprache verfügbar, die ebenfalls Listenverarbeitung unterstützt.

Dieser Aspekt der Listenverarbeitung ist allerdings bei Logo längst nicht so konsequent implementiert wie in Lisp, aber dafür auch für den Einsteiger recht schnell zu durchschauen. Bekannt geworden ist Logo allerdings in erster Linie durch die »Turtle-Grafik«, die schon zum Markenzeichen dieser Sprache geworden ist.

Die Turtle (Schildkröte) wird in der Regel durch ein kleines Dreieck am Bildschirm dargestellt, das mit Befehlen wie FORWARD, BACK, LEFT und RIGHT in alle Richtungen bewegt werden kann. Dabei hinterläßt die Schildkröte eine sichtbare Linie auf dem Bildschirm und kann so zum bequemen Zeichnen auch von komplexen Grafiken eingesetzt werden.

Wie bei Basic, so handelt es sich auch bei Logo (übrigens auch bei Lisp) um einen Interpreter, was ein sehr bequemes Vorgehen beim Programmieren erlaubt. Alle Programmbefehle können auch im Direktmodus eingesetzt werden, und so kann man alle Routinen direkt im Dialog mit dem Computer austesten, ohne das gesamte Programm wie bei einem Compiler nach jeder Änderung ständig wieder neu übersetzen zu müssen.

Die einfache Handhabung der Grafik ist sicherlich einer der Hauptgründe, die speziell beim C 64 für die Verwendung von Logo sprechen. Wenn Sie sich für Logo interessieren, dürfen wir Ihnen unseren Testbericht über das Commodore-Logo in dieser Ausgabe empfehlen.

Natürlich ist es mit den hier behandelten Sprachen noch nicht getan. Es existiert eine Unmenge weiterer Programmiersprachen, mit deren Behandlung man ganze Bücher füllen könnte. Dieser Bericht sollte Ihnen aber einen allgemeinen Überblick gegeben haben, wo's bei den Programmiersprachen lang geht.

Speziell für den C 64 wird es sicher in naher Zukunft eine Reihe weiterer Sprachen geben. Interessant wird die Angelegenheit natürlich auch im Hinblick auf den neuen Commodore 128 PC, der ja die ganze Welt der Programmiersprachen unter CP/M zugänglich macht. Ganz sicher werden wir das Thema »Programmiersprachen« unter diesem Aspekt nochmals aufgreifen. (ev)